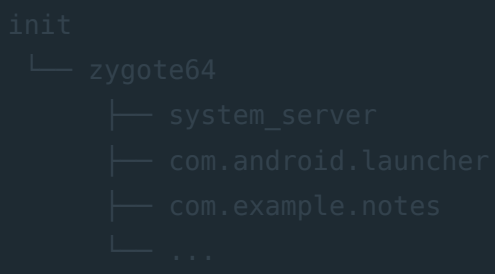
Android Zygote

A Complete Explanation

Process creation · fork() and copy-on-write · specialization

ART · system_server · ActivityThread · app startup



Technical Reference Guide

Contents

1. What is Zygote? . . . 5
2. Why does Android use Zygote? . . . 5
3. Android boot sequence . . . 6
4. How init starts Zygote . . . 6
    Important arguments . . . 7
5. From app_process to ZygoteInit . . . 7
6. What is ART? . . . 7
7. What happens inside ZygoteInit.main()? . . . 8
8. What does Zygote preload? . . . 9
    Framework classes . . . 9
    Framework resources . . . 9
    Native libraries and runtime infrastructure . . . 10
    Cleanup after preloading . . . 10
9. Why preloading saves memory . . . 10
10. Copy-on-write explained . . . 10
    Numerical example . . . 11
11. Complete application-launch example . . . 11
12. Who requests the new process? . . . 12
13. What information is sent to Zygote? . . . 12
14. Zygote command socket . . . 13
15. What happens when Zygote receives the command? . . . 14
16. Understanding the result of fork() . . . 14
17. What does "specialize" mean? . . . 15
18. UID-based application sandbox . . . 16
19. SELinux specialization . . . 16
20. Storage and mount namespaces . . . 17
21. Seccomp and syscall restrictions . . . 17
22. What happens after specialization? . . . 18
23. What is ActivityThread? . . . 18
24. The main thread and main Looper . . . 19
25. Binding the application . . . 19

26. When does your code begin? . . . 20
27. system_server . . . 21
28. Zygote versus system_server . . . 21
29. 32-bit and 64-bit Zygotes . . . 22
30. WebView Zygote . . . 22
31. App Zygote and child Zygote . . . 23
32. USAP: Unspecialized App Process . . . 23
33. Why Zygote controls thread creation carefully . . . 24
34. File descriptors and Zygote . . . 24
35. Cold, warm and hot application starts . . . 25
- Cold start . . . 25
- Warm start . . . 25
- Hot start . . . 25
36. Zygote does not remove all startup cost . . . 26
37. Zygote memory versus app memory . . . 26
- Commonly shared or shareable . . . 26
- Normally private to the app . . . 27
38. PSS, RSS and USS . . . 27
- RSS — Resident Set Size . . . 27
- PSS — Proportional Set Size . . . 27
- USS — Unique Set Size . . . 28
39. Do apps share Java objects? . . . 28
40. Does every Activity get a process? . . . 28
41. Practical ADB examples . . . 29
- See Zygote processes . . . 29
- Find an app process . . . 29
- Inspect process parent ID . . . 30
- Force a cold start . . . 30
- See startup timing in logcat . . . 30
- Inspect process memory . . . 31
- Trace startup with Perfetto . . . 31
42. Complete process tree example . . . 31
43. What happens if Zygote crashes? . . . 32

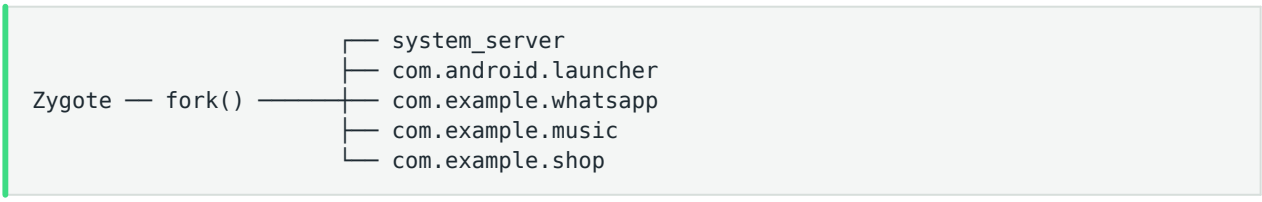
44. Common misconceptions 32
"Zygote is the Android virtual machine" 32
"Zygote is system_server" 32
"Every Activity is forked from Zygote" 32
"All apps have the same permissions because they come from Zygote" 33
"Fork immediately duplicates all physical memory" 33
"Applications can access each other's inherited objects" 33
"Zygote loads the application APK before forking every normal app" 33
"Zygote controls the entire app lifecycle" 33
45. End-to-end source-level call chain 33
46. Compact real-world example 34
47. Interview-ready answer 35
48. One-line summary 35

1. What is Zygote?

Zygote is a special Android system process that acts as the parent and prepared template for most Android application processes and for system_server.

Instead of initializing the Android Runtime and common framework components separately for every application, Android initializes them once inside Zygote. When a new process is required, Zygote creates it using a Linux process operation based on fork(). Android describes Zygote as the root of system and application processes using the same ABI.

The name comes from biology: a zygote is the original cell from which an organism develops. Android applications similarly begin as children of the Zygote process.



Zygote does not run every Android process. Native services such as some hardware, media, and low-level system daemons may be started directly by init rather than forked from Zygote.

2. Why does Android use Zygote?

Without Zygote, starting an application would require Android to repeat all of this:

- Create Linux process
- Initialize ART
- Initialize garbage collector
- Load core Java classes
- Load Android framework classes
- Load framework resources
- Initialize JNI
- Load common native libraries
- Configure the process
- Load the application

That would increase both startup time and memory consumption.

With Zygote:

```
Zygote initializes common runtime state once
                  ↓
        Android forks Zygote
                  ↓
Child inherits the prepared runtime state
                  ↓
Child becomes a specific application
```

Zygote therefore provides two main benefits:

- Faster application startup
- Memory sharing between application processes

3. Android boot sequence

A simplified Android boot process is:

```
Boot ROM
   ↓
Bootloader
   ↓
Linux kernel
   ↓
init process
   ↓
Zygote
   ↓
system_server
   ↓
Android system services
   ↓
Launcher and applications
```

The kernel starts Android's first userspace process, init. The init process reads Android initialization files, mounts filesystems, starts native services and launches Zygote. Zygote initializes the Java/ART environment, and system_server then starts core Android framework services.

4. How init starts Zygote

A current 64-bit AOSP Zygote service definition looks approximately like this:

```
service zygote /system/bin/app_process64 \
    -Xzygote \
    /system/bin \
    --zygote \
    --start-system-server \
    --socket-name=zygote
```

The service:

- Runs /system/bin/app_process64
- Starts it in Zygote mode

- Tells it to start system_server
- Gives the Zygote socket the name zygote
- Initially starts under the root identity
- Creates protected Zygote and USAP sockets

The current AOSP service configuration also assigns Zygote a high scheduling priority and marks it as a critical service.

Important arguments

-Xzygote — Configures the runtime for Zygote operation.

--zygote — Tells app_process that this is a Zygote process.

--start-system-server — Tells the primary Zygote to create system_server.

--socket-name=zygote — Specifies the Unix-domain socket used for Zygote commands.

5. From app_process to ZygoteInit

The native executable is commonly:

```
/system/bin/app_process64
```

or, on compatible configurations:

```
/system/bin/app_process32
```

The startup path is approximately:

```
init
  ↓
app_process64
  ↓
AndroidRuntime
  ↓
ART virtual machine
  ↓
com.android.internal.os.ZygoteInit.main()
```

The native app_process executable processes its command-line arguments and starts com.android.internal.os.ZygoteInit when running in Zygote mode.

6. What is ART?

ART means Android Runtime.

ART executes Android DEX bytecode and provides facilities such as:

- Class loading
- Garbage collection

- Thread management
- JNI integration
- Runtime verification
- Ahead-of-time compilation
- Just-in-time compilation
- Profile-guided optimization

ART is the managed runtime used by Android applications and some system services. It replaced Dalvik as Android's primary runtime, although both execute the DEX format.

Zygote and ART are not the same thing:

```
ART:
    Runtime that executes managed Android code

Zygote:
    Process containing a prepared ART runtime
    that creates application processes
```

7. What happens inside ZygoteInit.main()?

The main Zygote initialization flow is approximately:

```
ZygoteInit.main()
    ↓
Perform pre-fork runtime initialization
    ↓
Read ABI and socket arguments
    ↓
Preload common classes and resources
    ↓
Run garbage collection and finalization
    ↓
Initialize native Zygote state
    ↓
Create the Zygote server
    ↓
Fork system_server
    ↓
Enter command-processing loop
```

The current AOSP implementation preloads resources and classes, performs an initial cleanup, creates a ZygoteServer, optionally forks system_server, and then enters the Zygote socket loop.

Simplified pseudocode:

```
public static void main(String[] args) {
    preForkInitialization();

    parseArguments(args);

    preloadClassesResourcesAndLibraries();

    garbageCollectAfterPreload();

    ZygoteServer server = new ZygoteServer();

    if (startSystemServer) {
        forkSystemServer();
    }

    server.runSelectLoop();
}
```

This is explanatory pseudocode, not application code.

8. What does Zygote preload?

Zygote prepares data likely to be needed by many Android processes.

Framework classes

Examples conceptually include classes related to:

```
java.lang.*
java.util.*
android.os.*
android.app.*
android.content.*
android.view.*
android.graphics.*
```

For example:

```
java.lang.String
java.util.HashMap
android.app.Activity
android.content.Context
android.content.Intent
android.view.View
```

The exact preload list is controlled by the Android platform and can change between Android versions or device builds.

Framework resources

Zygote can preload common resources such as:

- Framework drawables
- Colors

- Resource tables
- Themes
- Common XML resources

Native libraries and runtime infrastructure

It also prepares selected native runtime and framework components that are useful across processes.

Cleanup after preloading

Preloading itself creates temporary objects. Zygote therefore performs garbage collection and finalization after the preload operation, before beginning normal process creation. The current source explicitly performs a post-Zygote-initialization GC after preloading.

9. Why preloading saves memory

The benefit comes from Linux virtual memory and copy-on-write.

Assume Zygote has loaded a framework class into memory:

```
Physical memory page containing framework data
                  ↑
             Zygote maps it
```

After applications are forked:

```
                     ┌── App A maps page
Physical page ───────┼── App B maps page
                     └── App C maps page
```

The applications can initially share the same physical page.

If all processes only read the page, the kernel does not need to create separate copies.

10. Copy-on-write explained

fork() creates a child process with a virtual-memory layout initially based on the parent.

Linux normally does not immediately copy every memory page. Instead, parent and child temporarily map the same physical pages.

Before write:

```
Physical Page X
   ↑       ↑
Zygote    App A
```

When App A modifies the page:

```
App A writes to Page X
        ↓
Kernel creates Page X2
```

Afterward:

```
Physical Page X  ← Zygote
Physical Page X2 ← App A
```

This mechanism is called copy-on-write, or COW.

Numerical example

Suppose a 4 KB framework page is used by three applications.

Without sharing:

```
App A: 4 KB
App B: 4 KB
App C: 4 KB

Physical memory: approximately 12 KB
```

With sharing:

```
App A ─┐
App B ─┼── one shared 4 KB physical page
App C ─┘

Physical memory: approximately 4 KB
```

Real Android memory accounting is more complicated because it includes clean, dirty, private, shared, anonymous, file-backed and compressed pages.

ART images containing commonly preloaded classes and objects can be shared across applications, and Android's memory tools report Zygote-related memory separately.

11. Complete application-launch example

Assume the user launches com.example.notes with com.example.notes.MainActivity.

The complete simplified flow is:

1. User taps the Notes icon.
2. Launcher sends an Activity launch request.
3. ActivityTaskManager receives the request.
4. Package and component information are resolved.
5. Android checks whether com.example.notes is running.
6. No process exists.
7. system_server prepares a process-start request.

8. The request is sent to the appropriate Zygote socket.
9. Zygote validates the request.
10. Zygote calls forkAndSpecialize().
11. The parent remains Zygote.
12. The child is configured as com.example.notes.
13. The child enters ActivityThread.main().
14. The child attaches to system_server using Binder.
15. Android binds the application to the process.
16. The APK class loader and app Context are prepared.
17. The Application instance is created.
18. App providers and application initialization run.
19. Application.onCreate() executes.
20. MainActivity is created.
21. MainActivity.onCreate() executes.
22. Views are inflated, measured, laid out and drawn.
23. The first frame replaces the starting window.

A cold start includes process creation, application-object creation, main-thread startup, Activity creation, view inflation, layout and initial drawing.

12. Who requests the new process?

The application itself normally does not communicate with Zygote directly.

High-level system services running inside system_server decide that a new application process is necessary.

The conceptual path is:

```
Launcher
   ↓ Binder
ActivityTaskManagerService
   ↓
ActivityManager / process management
   ↓
Process start request
   ↓
ZygoteProcess
   ↓ Unix-domain socket
Zygote
```

ZygoteProcess maintains addresses for the primary Zygote, secondary Zygote and corresponding USAP sockets. It opens a local socket connection and exchanges process-start commands with Zygote.

13. What information is sent to Zygote?

A process-start command can contain information such as:

- Entry-point class
- Process name
- UID
- GID
- Supplementary groups
- Runtime flags
- Target SDK version
- SELinux information
- ABI
- Instruction set
- Application data directory
- Mount configuration
- Package name
- Debug configuration
- Compatibility changes
- Whether this is a top application
- Storage directory information

The current ZygoteProcess.startViaZygote() interface includes values such as UID, GID, groups, runtime flags, target SDK, SELinux information, ABI, instruction set, app-data directory, package name and mount-related options.

Conceptually:

```
start-process
uid=10234
gid=10234
nice-name=com.example.notes
target-sdk=...
seinfo=default
abi=arm64-v8a
entry-class=android.app.ActivityThread
app-data-dir=/data/user/0/com.example.notes
```

This is a conceptual representation, not the exact stable socket protocol.

14. Zygote command socket

Zygote listens on a protected Unix-domain socket.

A Unix-domain socket is an operating-system IPC channel between processes on the same device. It is not an internet socket.

The Zygote server loop:

```
Wait for socket activity
        ↓
Accept connection
        ↓
Read one process command
        ↓
Validate command
        ↓
Fork or select a USAP
        ↓
Return new PID
```

The AOSP ZygoteServer.runSelectLoop() method continuously monitors the server and client sockets and reads process requests.

Ordinary third-party applications cannot freely use Zygote to create arbitrary privileged processes. Socket permissions, caller credentials and Zygote security checks restrict what can be requested.

15. What happens when Zygote receives the command?

The request is processed by code such as ZygoteConnection.processCommand().

The simplified operation is:

```
arguments = readAndParseCommand();

checkUidPolicy(arguments);
checkCapabilities(arguments);
checkDebuggerPolicy(arguments);

pid = Zygote.forkAndSpecialize(...);

if (pid == 0) {
    // Child
    closeZygoteSockets();
    return startApplication(arguments);
} else {
    // Parent Zygote
    reportChildPid(pid);
}
```

The actual implementation validates capabilities and UID-related policies, closes inherited Zygote descriptors in the child, calls forkAndSpecialize(), follows the child path when the returned PID is zero, and follows the parent path otherwise.

16. Understanding the result of fork()

Conceptually:

```
pid_t pid = fork();

if (pid < 0) {
    // Failure
} else if (pid == 0) {
    // Child process
} else {
    // Parent process
    // pid is the child's process ID
}
```

For Zygote:

```
Parent branch:
    Continue as Zygote
    Keep listening for more commands
    Report child PID

Child branch:
    Stop behaving as Zygote
    Close Zygote server descriptors
    Apply application security identity
    Start Android application entry point
```

Zygote itself does not permanently turn into the application. It remains alive and can create more processes.

17. What does "specialize" mean?

Immediately after the fork, the child is still too similar to the privileged Zygote process.

Android must transform it into a restricted process belonging to one specific app. This is called specialization.

Conceptually:

```
Generic Zygote child
        ↓
Set application UID and GID
        ↓
Set supplementary groups
        ↓
Remove unnecessary capabilities
        ↓
Apply SELinux process context
        ↓
Create storage/mount view
        ↓
Apply seccomp restrictions
        ↓
Set resource limits
        ↓
Set process name
        ↓
Apply runtime/debug flags
        ↓
Start app entry point
```

The current native Zygote implementation exposes separate native operations for forking and specializing normal app processes, system_server, USAP processes and child Zygotes.

18. UID-based application sandbox

Android normally assigns each application its own Linux UID.

Example:

```
com.example.notes → UID 10234
com.example.bank  → UID 10235
com.example.game  → UID 10236
```

Linux permission checking then separates their files and resources.

For example:

```
/data/user/0/com.example.notes/
/data/user/0/com.example.bank/
```

The Notes process should not be able to freely read the Bank app's private directory.

Android's application sandbox uses Linux user-based isolation, with applications running under separate identities. SELinux, filesystem controls, seccomp and additional platform mechanisms strengthen that isolation.

19. SELinux specialization

Android uses SELinux mandatory access control in addition to ordinary Linux UID permissions.

A child process receives a security domain appropriate to its role.

Conceptually:

```
Zygote domain
     ↓ fork + transition
Untrusted application domain
```

A system application, isolated process, regular third-party app and system_server do not necessarily receive the same SELinux domain or privileges.

SELinux has been used to strengthen Android application sandbox boundaries, and modern Android runs it in enforcing mode.

20. Storage and mount namespaces

A process can receive a restricted filesystem view during specialization.

A mount namespace lets different processes observe different filesystem mount layouts, even when they run on the same kernel.

Conceptually:

```
App A sees:
    its own app data
    permitted shared storage locations

App B sees:
    its own app data
    permitted shared storage locations
```

Android can hide unrelated application data directories and bind-mount only the permitted directories into a private mount namespace. Current Zygote native code contains app-data isolation logic for this purpose.

21. Seccomp and syscall restrictions

Linux system calls are direct requests to the kernel, such as:

```
open
read
write
mmap
socket
ioctl
clone
```

Android can install seccomp-BPF filters that limit which system calls an application process may use.

Android's sandbox documentation states that application processes use syscall filtering to strengthen the application/kernel boundary.

Therefore, specialization is not just "change process name" — it is an important security transition.

22. What happens after specialization?

The child proceeds through Android runtime initialization.

The path is approximately:

```
Zygote child
   ↓
ZygoteInit.zygoteInit()
   ↓
RuntimeInit.commonInit()
   ↓
Start Binder thread-pool integration
   ↓
RuntimeInit.applicationInit()
   ↓
Find static main() entry point
   ↓
ActivityThread.main()
```

Current AOSP code shows zygoteInit() performing common runtime initialization, native Zygote child initialization and application-entry-point resolution.

For a normal application, the entry point is effectively android.app.ActivityThread.main().

23. What is ActivityThread?

ActivityThread is a central internal controller for an Android application process.

Despite its name, it is not simply a normal Thread object that represents an Activity.

Its responsibilities include coordinating:

- Application binding
- Activity lifecycle commands
- Service creation
- Broadcast receiver execution
- Content provider installation
- Configuration changes
- Communication with system_server
- The application's main message loop

The current ActivityThread.main():

- Performs per-process initialization.
- Prepares the main Looper.
- Creates an ActivityThread.

- Attaches it to the system.
- Enters Looper.loop().

Simplified:

```
public static void main(String[] args) {
    Looper.prepareMainLooper();

    ActivityThread thread = new ActivityThread();
    thread.attach(false);

    Looper.loop();
}
```

24. The main thread and main Looper

By default, an Android app begins with one main thread.

That thread:

- Processes lifecycle messages
- Handles most component callbacks
- Dispatches input
- Updates Views
- Performs UI drawing coordination

Android documents that a new application process begins with a main thread and that application components run in the same process and main thread by default.

The main Looper behaves conceptually like this:

```
while (true) {
    Message message = queue.next();
    message.target.dispatchMessage(message);
}
```

This is why expensive work on the main thread causes freezes or ANRs.

25. Binding the application

After ActivityThread attaches, system_server sends the application configuration into the process.

The process then receives information including:

- ApplicationInfo
- Package details
- Class-loader information
- Instrumentation configuration
- Providers

- Compatibility data
- Runtime configuration
- Process name
- User and storage information

The relevant internal operation is commonly associated with ActivityThread.handleBindApplication().

Current AOSP routes the BIND_APPLICATION main-thread message to handleBindApplication().

26. When does your code begin?

A developer may think process startup begins at Application.onCreate(). But much has already happened:

```
Zygote startup
Framework preload
Process request
fork()
Process specialization
ART child initialization
ActivityThread.main()
Main Looper initialization
Binder attachment
Application binding
Class-loader creation
Application instance construction
Application.onCreate()
```

Your custom Application code might be:

```
class NotesApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        // Your application initialization starts here.
    }
}
```

After this, Android creates the requested component, for example:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Application.onCreate() is therefore not process creation. It is a callback inside an already-created and specialized process.

27. system_server

One of Zygote's earliest and most important children is system_server.

Zygote uses a separate system-server fork path: Zygote.forkSystemServer(...).

The parent remains Zygote. The child continues as system_server. This behavior is visible in the current ZygoteInit implementation.

system_server hosts many Android framework services, including services responsible for areas such as:

- Activity and task management
- Package management
- Window management
- Power management
- Notifications
- Jobs
- Input methods
- Device policy
- User management
- Connectivity coordination

Simplified relationship:

```
Zygote
  ├── system_server
  │     ├── ActivityTaskManagerService
  │     ├── ActivityManagerService
  │     ├── PackageManagerService
  │     ├── WindowManagerService
  │     └── many other services
  │
  ├── launcher process
  ├── settings process
  └── application processes
```

system_server decides and coordinates much of Android's high-level system behavior. Zygote primarily supplies the mechanism for creating prepared runtime processes.

28. Zygote versus system_server

Zygote	system_server
Creates ART-based processes	Runs framework system services
Preloads common runtime state	Manages activities, packages, windows and more
Waits on a process-command socket	Communicates heavily through Binder

Zygote	system_server
Forks application children	Requests application process creation
Parent of system_server	Child of Zygote

A useful summary is:

- system_server decides that an app process is needed.
- Zygote creates the process.
- The kernel schedules and isolates it.
- ART executes its managed code.

29. 32-bit and 64-bit Zygotes

An Android device can support different application binary interfaces:

```
arm64-v8a
armeabi-v7a
x86_64
x86
```

On a dual-architecture device there may be zygote64 and zygote32, or primary and secondary Zygotes.

A process must originate from a Zygote compatible with its ABI.

```
zygote64
   ├── 64-bit system_server
   ├── 64-bit App A
   └── 64-bit App B

zygote32
   ├── 32-bit App C
   └── 32-bit App D
```

Current Android documentation notes that dual-architecture systems can start separate 32-bit and 64-bit Zygote processes. It also notes that modern devices such as Pixel 7 and later use a 64-bit main Zygote.

Not every device has two Zygotes. Many modern devices are 64-bit-only.

30. WebView Zygote

Android can use a specialized WebView Zygote.

It is prepared with components useful to processes that execute WebView or Chromium-related code.

```
Main Zygote
   → normal application processes

WebView Zygote
   → selected WebView renderer processes
```

This allows WebView-related child processes to inherit WebView-specific libraries and resources rather than loading everything independently.

Current Android documentation identifies WebView Zygote as a specialized Zygote containing libraries and resources for WebView processes.

31. App Zygote and child Zygote

Android can create a child Zygote for specialized process groups.

Conceptually:

```
Main Zygote
     ↓
App-specific child Zygote
     ├── isolated child process 1
     ├── isolated child process 2
     └── isolated child process 3
```

This is useful for cases in which multiple restricted processes need the same application-specific code or resources.

AOSP's ZygoteProcess process-start interface explicitly supports creating a child Zygote whose state is cloned from the parent Zygote.

This is an advanced platform feature and is not the ordinary path for every Activity.

32. USAP: Unspecialized App Process

Modern Android can optionally maintain an Unspecialized App Process pool.

USAP means Unspecialized App Process.

A USAP has already been forked from Zygote but has not yet been assigned to a specific app.

```
Zygote
   ├── Generic USAP 1
   ├── Generic USAP 2
   └── Generic USAP 3
```

When an application needs a process:

```
Available USAP
    ↓
Apply app UID/GID
    ↓
Apply SELinux and mount configuration
    ↓
Assign process name
    ↓
Start application
```

This can remove the fork operation from a latency-sensitive launch path.

Current Android documentation says Zygote can either fork on demand or create a USAP pool. Once a USAP is assigned to an application, it leaves the pool, and Zygote replenishes the pool when necessary.

Not every process-start request is eligible for a USAP. Current AOSP policy generally considers latency-sensitive, non-system process launches, and certain command options cannot be sent to a USAP.

33. Why Zygote controls thread creation carefully

Forking a multithreaded process is dangerous.

Example:

```
Thread A acquires Lock X
Thread B calls fork()
```

After the fork, the child initially continues with only the calling thread, but copied runtime state might show Lock X as owned by Thread A.

```
Child:
    Thread B exists
    Lock X looks locked
    Thread A does not exist
```

The child could deadlock forever.

Zygote and ART therefore carefully coordinate runtime state and thread behavior around fork operations. The current Zygote entry point explicitly uses runtime hooks that prohibit thread creation during sensitive initialization stages.

This is one reason developers should not think of Zygote as simply "fork();" — the platform performs significant runtime preparation before and after the native fork.

34. File descriptors and Zygote

A child created by fork() can inherit open file descriptors from its parent.

That can be dangerous because a child might inherit access to:

- Server sockets
- Pipes
- Open files
- Devices
- Internal system communication channels

Zygote therefore tracks and closes descriptors that should not be inherited.

For example, the application child must not retain Zygote's command socket. The AOSP command-processing path explicitly arranges for Zygote socket descriptors to be closed in the child before it transitions to the application's UID and privileges.

35. Cold, warm and hot application starts

Cold start

The application process does not exist:

```
Request process
   ↓
Zygote fork or USAP selection
   ↓
Specialization
   ↓
ActivityThread
   ↓
Application creation
   ↓
Activity creation
   ↓
First frame
```

Zygote is directly involved.

Warm start

The process may exist, but the Activity needs recreation:

```
Existing application process
   ↓
Create or restore Activity
```

Normally no new Zygote fork is required.

Hot start

The application process and Activity already exist:

```
Bring existing Activity to foreground
```

No new process is needed.

Android officially distinguishes cold, warm and hot starts. A cold start requires the most work because the system must create the application process and the application must initialize its objects and first Activity.

36. Zygote does not remove all startup cost

Zygote makes process creation more efficient, but your app can still have a slow cold start.

Bad example:

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        loadLargeDatabaseSynchronously()
        readHugeJsonFile()
        initializeEverySdk()
        decodeLargeImages()
        performBlockingNetworkRequest()
    }
}
```

These operations happen after the process has been created and can delay the first frame.

Better:

```
class MyApplication : Application() {
    override fun onCreate() {
        super.onCreate()

        initializeOnlyCriticalComponents()
        scheduleNonCriticalInitialization()
    }
}
```

Application and Activity creation are common sources of startup overhead, especially expensive work in Application.onCreate() and Activity onCreate().

37. Zygote memory versus app memory

Memory inherited from Zygote can remain shared, but each application also allocates private data.

Commonly shared or shareable

- Framework executable code
- ART image pages
- Preloaded class data
- Read-only native library code
- Clean memory-mapped DEX/OAT pages

- Read-only resources

Normally private to the app

- Activity objects
- View hierarchy
- ViewModels
- Application-specific caches
- Mutable app objects
- Thread stacks
- App-specific bitmaps
- Modified inherited pages
- Native allocations

Example:

```
Shared between apps:
    android.app.Activity implementation
    java.lang.String implementation
    ART boot image

Private to Notes:
    MainActivity instance
    NoteRepository
    Notes database cache

Private to Bank:
    LoginActivity instance
    Session state
    Account data objects
```

38. PSS, RSS and USS

Because Zygote descendants share memory, process memory cannot be understood by looking only at one simple number.

RSS — Resident Set Size

All resident physical pages mapped by the process, including shared pages.

If three apps share the same 10 MB, each app's RSS can include that 10 MB. Adding their RSS values would count the same physical pages multiple times.

PSS — Proportional Set Size

Shared memory is divided proportionally between the processes using it.

Example:

```
Shared page group: 12 MB
Processes sharing it: 3

Each process receives approximately 4 MB PSS
```

USS — Unique Set Size

Memory private to one process.

Android's dumpsys meminfo documentation describes total PSS as the process's proportional memory weight and distinguishes private clean and private dirty allocations from shared memory.

39. Do apps share Java objects?

No.

Applications may initially map some of the same physical memory pages, but every process has its own virtual address space.

Suppose App A and App B inherit an object from Zygote:

```
App A changes object
```

Copy-on-write gives App A a private modified page.

App B does not see App A's normal heap change.

Apps communicate using explicit IPC mechanisms such as:

- Binder
- Bound services
- Intents
- Content providers
- Sockets
- Carefully designed shared memory
- Files where access is permitted

Being children of the same process does not produce a shared Java heap.

40. Does every Activity get a process?

No.

By default, all components in one application run in the same process:

```
com.example.notes process
   ├── MainActivity
   ├── EditActivity
   ├── SyncService
   ├── BootReceiver
   └── NotesProvider
```

Starting a second Activity normally does not require Zygote to create another process.

Android creates a process when an application component needs to run and no suitable process already exists. Components use the same application process and main thread by default, although the manifest can assign components to different processes.

Example separate process:

```
<service
    android:name=".UploadService"
    android:process=":upload" />
```

This may create:

```
com.example.notes
com.example.notes:upload
```

Each is a different Linux process, and each process may require its own Zygote-originated startup.

41. Practical ADB examples

See Zygote processes

```
adb shell ps -A | grep zygote
```

Possible output:

```
root       512     1  ...  zygote64
webview_zygote  811   1  ...  webview_zygote
```

A dual-ABI device might also show:

```
zygote
zygote64
```

Exact names and output vary by Android version and manufacturer.

Find an app process

```
adb shell ps -A | grep com.example.notes
```

Example:

```
u0_a234   12450   512  ...  com.example.notes
```

Here:

```
12450 = application PID
512   = parent PID
```

If 512 is Zygote, the output demonstrates the parent-child relationship.

On some devices or process-list implementations, parent information may be displayed differently.

Inspect process parent ID

```
adb shell cat /proc/12450/status | grep PPid
```

Possible result:

```
PPid:   512
```

Then inspect PID 512:

```
adb shell cat /proc/512/cmdline
```

Possible result:

```
zygote64
```

Access to some /proc information can be restricted on production devices.

Force a cold start

```
adb shell am force-stop com.example.notes
adb shell am start -W com.example.notes/.MainActivity
```

Possible result:

```
Status: ok
LaunchState: COLD
Activity: com.example.notes/.MainActivity
TotalTime: 620
WaitTime: 635
Complete
```

Output fields vary by Android version.

See startup timing in logcat

```
adb logcat | grep "Displayed"
```

Example:

```
ActivityTaskManager: Displayed com.example.notes/.MainActivity: +620ms
```

Android's startup documentation identifies the Displayed log entry as one way to retrieve time to initial display.

Inspect process memory

```
adb shell dumpsys meminfo com.example.notes
```

Useful categories may include:

```
Native Heap
Dalvik Heap
Dalvik Other
Stack
Ashmem
.so mmap
.dex mmap
.art mmap
.Zygote
TOTAL PSS
Private Dirty
Private Clean
```

The exact report depends on the Android version, build type and command options.

Trace startup with Perfetto

Perfetto can display:

- Android App Startups
- Process creation
- Main thread work
- Binder calls
- Application initialization
- Activity lifecycle
- First-frame timing

Android's current startup guidance recommends inspecting the Android App Startups derived metric and correlating it with the application's main-thread trace.

42. Complete process tree example

A simplified process tree could look like:

```
init
├── servicemanager
├── surfaceflinger
├── audioserver
├── cameraserver
├── zygote64
│   ├── system_server
│   ├── com.android.systemui
│   ├── com.android.launcher
│   ├── com.android.settings
│   ├── com.example.notes
│   └── com.example.bank
└── webview_zygote
    ├── WebView renderer 1
    └── WebView renderer 2
```

This is illustrative. Actual Android process trees depend on the device, architecture, OS version, OEM and active applications.

43. What happens if Zygote crashes?

Zygote is critical to Android operation.

If it dies:

- New normal application processes cannot be created.
- system_server may also be affected or restarted.
- Android's init process detects the service failure.
- The system may restart critical framework processes or reboot, depending on failure policy and build configuration.

The current Zygote service is marked critical, and its configuration includes restarting several dependent services if Zygote restarts.

A normal third-party application crash does not kill Zygote because the application is a separate process.

44. Common misconceptions

"Zygote is the Android virtual machine"

Incorrect.

```
ART = runtime
Zygote = process containing prepared ART state
```

"Zygote is system_server"

Incorrect. Zygote forks system_server. They continue as different processes.

"Every Activity is forked from Zygote"

Incorrect. Zygote creates processes, not individual Activity objects. Multiple Activities commonly run in one application process.

"All apps have the same permissions because they come from Zygote"

Incorrect. Children are specialized with different UIDs, GIDs, SELinux contexts, capabilities, mount views and runtime restrictions.

"Fork immediately duplicates all physical memory"

Incorrect. Copy-on-write allows pages to remain shared until one process modifies them.

"Applications can access each other's inherited objects"

Incorrect. They have separate virtual address spaces.

"Zygote loads the application APK before forking every normal app"

Incorrect. Zygote preloads common platform state. After specialization, the child loads and binds the specific application's code and resources. Specialized child-Zygote and WebView cases provide additional mechanisms.

"Zygote controls the entire app lifecycle"

Incorrect. system_server services coordinate component and process lifecycle policy. Zygote supplies process creation.

45. End-to-end source-level call chain

A useful conceptual call chain is:

```
init
 └── app_process64
      └── AndroidRuntime.start()
           └── ZygoteInit.main()
                ├── preload()
                ├── gcAndFinalize()
                ├── forkSystemServer()
                └── ZygoteServer.runSelectLoop()
```

When an app is needed:

```
ActivityTaskManager / ActivityManager
 └── process-start logic
      └── ZygoteProcess.start(...)
           └── startViaZygote(...)
                └── Unix-domain socket command
                     └── ZygoteConnection.processCommand()
                          └── Zygote.forkAndSpecialize()
                               └── nativeForkAndSpecialize()
```

Child branch:

```
handleChildProc()
 └── ZygoteInit.zygoteInit()
      └── RuntimeInit.applicationInit()
           └── ActivityThread.main()
                ├── Looper.prepareMainLooper()
                ├── ActivityThread.attach()
                └── Looper.loop()
```

Application binding:

```
system_server
 └── bind application through Binder
      └── ActivityThread.handleBindApplication()
           ├── configure class loader and Context
           ├── create Application
           ├── initialize providers
           ├── call Application.onCreate()
           └── create requested component
```

The exact internal method chain changes as Android evolves, but this model accurately captures the architecture represented by current AOSP Zygote, ZygoteServer, ZygoteConnection and ActivityThread code.

46. Compact real-world example

Suppose the device already booted and the user taps a calculator application:

```
Package: com.example.calculator
Process: com.example.calculator
UID: 10241
ABI: arm64-v8a
```

Android performs this sequence:

```
Launcher sends launch request
        ↓
system_server resolves CalculatorActivity
        ↓
No calculator process is running
        ↓
system_server sends request to zygote64
        ↓
zygote64 forks or assigns a USAP
        ↓
Child gets UID 10241
        ↓
Child gets calculator SELinux context
        ↓
Child receives calculator storage view
        ↓
Child process name becomes com.example.calculator
        ↓
ActivityThread.main() begins
        ↓
Application.onCreate()
        ↓
CalculatorActivity.onCreate()
        ↓
Layout is inflated
        ↓
First frame appears
```

The Zygote parent remains zygote64 and can later create:

```
com.example.camera
com.example.music
com.example.browser
```

47. Interview-ready answer

A strong interview response would be:

Zygote is a long-running Android system process started by init. It initializes ART and preloads common Android framework classes, resources and runtime state. When system_server needs an application process, it sends a request to Zygote through a protected Unix-domain socket. Zygote forks itself, and Linux copy-on-write allows the child to share unchanged memory pages with Zygote and other children. The child is then specialized with the app's UID, GID, SELinux context, mount namespace, process name and runtime restrictions. Finally, it starts ActivityThread.main(), attaches to the framework, creates the Application, and launches the requested component. This improves startup performance and reduces total memory consumption.

48. One-line summary

Zygote = a preinitialized ART process that efficiently creates secure Android app processes using fork, copy-on-write and process specialization.